RAMCube: Exploiting Network Proximity for RAM-Based Key-Value Store

Yiming Zhang†, Chuanxiong Guo‡, Rui Chu†, Guohan Lu‡, Yongqiang Xiong‡, Haitao Wu‡
†National University of Defense Technology
‡Microsoft Research Asia
{ymzhang, rchu}@nudt.edu.cn, {chguo,lguohan,yqx, hwu}@microsoft.com

Abstract

Disk-based storage is becoming increasingly problematic in meeting the needs of large-scale cloud applications. Recently RAM-based storage is proposed by aggregating the RAM of thousands of commodity servers in data center networks (DCN). These studies focus on improving performance with low latency RPC and fast failure recovery. RAM-based storage brings great DCN-related challenges, e.g., false server failure detection due to network problems, traffic congestion during failure recovery, and ToR switch failure handling.

This paper presents *RAMCube*, a DCN-oriented design for RAM-based key-value store based on the BCube network [9]. RAMCube exploits the properties of BCube to restrict all failure detection and recovery traffic within one-hop neighborhood, and leverages BCube's multiple paths to handle switch failures. Prototype implementation demonstrates that RAMCube is promising to achieve high performance I/O and fast failure recovery in large-scale DCNs.

1 Introduction

Disk-based storage is becoming more and more problematic in meeting the needs of large-scale cloud systems in terms of I/O latency and bandwidth. As a result, in recent years we see an increasing trend of migration of data from disks to random access memory (RAM) in storage systems. For example, memcached [1] is an in-memory key-value store that has been widely used by a number of Web service providers, including Facebook, Twitter, and Youtube, to offload their storage servers. PIBUS [13] aggregates the RAM of huge number of nodes on the Internet to act as a remote disk cache of desktop I/O-intensive applications. Google and Microsoft keep entire search indexes in RAM [11], and Google's Bigtable keeps certain columns (or even entire column family) in RAM [4].

Keeping data in RAM brings great challenges to reliable data access. Cache-based approaches (like memcached) cause difficulties for applications to effectively utilize RAM. For example, it is the responsibility of applications to manage consistency between caches and disk-based storage, making it vulnerable to consistency problems. Most recently, RAMCloud [11] is proposed as a RAM-based key-value store where data is kept entirely in the RAM of thousands of servers. It achieves fast server failure recovery by scattering backup data across a large number of disks and reconstructing lost data in parallel across expensive InfiniBand networks [4, 8].

RAM-based storage (like RAMCloud) has a number of benefits such as low latency and high bandwidth/throughput. Moreover, the applications need no longer manage the consistency between RAM and a separate backing store. To achieve reliable RAM-based storage in data centers, however, many realistic DCN-related issues need to be addressed. (i) It is difficult to quickly (e.g., in 100 ms) distinguish temporary network problems (e.g., temporary network partition or packet loss at some congested switches) from server failures across a large-scale network, which may consequently cause inconsistency after failure recovery. (ii) The large number (up to thousands) of parallel unarranged recovery flows is likely to bring traffic congestion, resulting in unexpected recovery delay. (iii) Top-of-rack (ToR) switch failures (with all servers in the rack still being alive) brings great difficulty to fast failure recovery.

This work describes *RAMCube*, a DCN-oriented design for RAM-based key-value store that supports thousands or tens of thousands of servers to offer up to hundreds of terabytes of RAM storage. In this paper, we follow the technical trend that large data centers are constructed using commodity Ethernet switches, and use Ethernet-based BCube [9], which is a server-centric network, as the underlying network of our RAMCube. RAMCube exploits the proximity of BCube network to construct a symmetric *MultiRing* structure, restricting all failure detection and recovery traffic within

one-hop neighborhood, which addresses the aforementioned problems including false failure detection and recovery traffic congestion. In addition, RAMCube leverages BCube's multiple paths between any pairs of servers to handle switch failures.

The rest of the paper is organized as follows. Section 2 discusses background and related work. Section 3 presents RAMCube design. Section 4 introduces prototype implementation and experiments. Finally, Section 5 concludes the paper.

2 Background and Related Work

2.1 RAM-Based Storage

The idea of permanently storing data in RAM is not new. For example, main-memory databases [7] keep entire databases in the RAM of one or more servers and support full RDBMS semantics. However, these systems cannot survive coordinated server failures and do not provide enough durability for large-scale systems.

Most recently, RAMCloud [11] is proposed as a RAM-based key-value store in data centers, where data (key-value pair) is kept entirely in RAM and large-scale systems are created by aggregating the RAM of thousands of commodity servers. RAMCloud realizes low-latency RPC by using expensive InfiniBand networks and supporting user-level applications to send/receive data directly through the NICs (bypassing the kernel).

RAMCloud keeps only one single copy of each object in a master server's RAM, with redundant copies in backup servers' disks. RAMCloud realizes fast server failure recovery by using aggressive data partitioning, a distributed approach similar to Google's Bigtable [4] and GFS [8]. They scatter backup data across hundreds or thousands of disks on backup servers, and reconstruct lost data in the RAM of hundreds of servers in a short period of time. RAMCloud employs randomized techniques, mainly including random replica placement (with refinement) for load balance and random ping for server failure detection, to manage the system in a decentralized and scalable fashion.

In this paper, we improve RAMCloud by addressing several critical issues including false failure detection of servers due to temporary network problems, traffic congestion during failure recovery, and ToR switch failure handling, by leveraging the properties of BCube.

2.2 BCube

BCube [9] is a server-centric network architecture designed for modular data centers. In BCube, servers with multiple network ports connect to multiple layers of COTS (commodity of the-shelf) mini-switches.

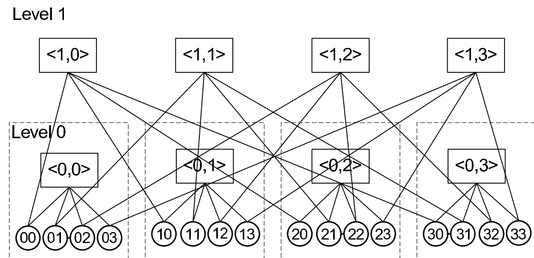

Figure 1: An example of BCube$(4,1)$ [9].

Servers act as not only end hosts, but also relay nodes for each other. BCube supports various bandwidth-intensive applications by speeding-up one-to-one, one-to-several, and one-to-all traffic patterns, and exhibits graceful performance degradation as the server and/or switch fail. This property is of special importance for fast failure recovery in RAM-based storage systems, since once a routing component fails it is very difficult to recover all of its connected servers in a short period of time.

BCube is a recursively defined structure. A BCube$(n,0)$ is simply n servers connecting to an n-port switch. A BCube$(n,1)$ is constructed from n BCube$(n,0)$ and n n-port switches. More generically, a BCube(n,k) ($k \geq 1$) is constructed from n BCube$(n,k-1)$ and n^k n-port switches. Each server in a BCube(n,k) has $k+1$ ports, which are numbered from level-0 to level-k. BCube(n,k) has $N = n^{k+1}$ servers and $k+1$ levels of switches, with each level having n^k n-port switches. Fig. 1 shows an example of BCube$(4,1)$, which is constructed from four BCube$(4,0)$ and four 4-port switches.

BCube's software-based routing approach suffers from high CPU overhead and processing latency. To address this problem, most recently Lu et al. design and implement *ServerSwitch* [10], a programmable commodity switching chip that supports high-performance BCube routing and achieves very low CPU overhead, high throughput and low processing latency.

3 RAMCube Design

3.1 Basics

We first briefly discuss some basic choices of RAMCube in network hardware, data model, and structure.

Network hardware. Network hardware is an important factor that decides the performance a RAM-based storage system can achieve. Infiniband is featured by its high bandwidth, low latency, as well as high price. For example, in a small-scale (60-server) InfiniBand testbed, RAMCloud claims 400∼800 MB/s recovery bandwidth per NIC and an RPC latency of 5∼10 μs [11]. However, our view is that high-performance Ethernet is more promising and cost-effective than InfiniBand for large

data centers. Recent technology trends show that Ethernet switches with 40 Gbit/s bandwidth and sub-μs latency are practical in the near future. We therefore design RAMCube based on Ethernet hardwares (BCube [9] with ServerSwitch [10] support).

Data model. The current data model in RAMCube is a simple key-value store that supports arbitrary number of tables containing key-value pairs. A key-value pair consists of a variable-length (up to 1 KB) key and a variable-length (up to 1 MB) value. RAMCube provides a simple set of operations ("*set* key value", "*get* key") and "*delete* key" for writing/updating, reading and deleting data. In the future RAMCube will extend the data model with support of more powerful features such as indexes and super columns [4].

Primary-recovery-backup. RAMCube stores multiple copies for each key-value pair for durability. There are two choices, namely *symmetric replication* [5] and *primary-backup* [3], for RAMCube to maintain consistency in normal read/write operations. In symmetric replication all copies of a key-value pair have to be kept in the RAM of different servers and a quorum-like technique [6] is used for conflict resolution. In contrast, in primary-backup only one primary copy is needed to be stored in RAM with redundant backup copies in disks. All read/write operations are through the primary copy. Clearly, for RAMCube primary-backup is preferred since it saves much of the RAM compared to symmetric replication.

We refer to the servers storing the primary copies in RAM as *primary servers*, and the servers storing the backup copies in the disks as *backup servers*. Considering the typical bandwidth of disks (100~200 MB/s), if we want to recover a primary server failure in a short period of time (1~2 seconds), one primary server with 64 GB RAM needs at least several hundred backup disks. Once having been read from disks of backup servers, the backup data should be recovered to as *few* as possible healthy servers since fragmentation changes the locality of original data and might degrade application performance after recovery. The healthy servers that accommodate the recovered data are called *recovery servers*. Considering the typical NIC bandwidth (10 Gbps), at least tens of recovery servers are needed for recovering a failed primary server with 64 GB RAM in 1~2 seconds. This "primary-recovery-backup" structure [4, 11] is depicted in Fig. 2(a). Note that each server symmetrically acts as all the three roles at the same time.

3.2 RAMCube Structure

Fast failure recovery is crucial for RAMCube to improve availability. The problem is that primary-backup is not tolerant of false server failure recovery that may result in two primary servers for the same key, while various temporary network problems (e.g., timeout due to network congestion) are difficult to be quickly distinguished from real server failures in large-scale networks.

The basic idea of RAMCube for addressing this problem is to utilize the global topology information of BCube and leverage network *proximity* to restrict all failure detection and recovery traffic within one-hop neighborhood. We improve the primary-recovery-backup structure (shown in Fig. 2(a)) with a *directly* connected tree (shown in Fig. 2(b)), where a primary server has multiple directly connected recovery servers, each of which corresponding to multiple directly connected backup servers. Clearly, Fig. 2(b) can be viewed as a special case of Fig. 2(a).

The primary server periodically sends heartbeat messages to all its recovery servers, and once the recovery servers detect (with certain mechanisms described in the next subsection) the primary server fails, they will start a recovery procedure reading backup data from their directly connected backup servers. In Fig. 2(b), since the recovery servers directly connect to the primary server, they can eliminate much of the possibility of false failure detection. In addition, since the recovery servers directly connect to the backup servers, the recovery traffic is guaranteed to have little overlap or congestion.

The directly connected tree provides great benefit for accurate failure detection and fast failure recovery. In order to apply it to RAMCube, we need symmetrically map the tree onto the entire network, that is, each server equally plays all the three roles of primary server, recovery server and backup server. Our insight is that for BCube if we replace each switch and its n links with an $n \times (n-1)$ full mesh that directly connects the servers, we will get a generalized Hypercube [2]. Then, we can construct multi-layer logical rings (*MultiRing* for short) for symmetric mapping as depicted in Fig. 3.

(1) The first layer ring is called *primary ring*, which is composed of all servers in BCube. The whole key space is mapped onto the primary ring and each primary server is responsible for a subset of the key space.

(2) Each primary server, say server P, has a second layer ring called *recovery ring* that is composed of all one-hop neighbors of P. When P fails, its data should be recovered to the RAM of the recovery servers on its recovery ring. Fig. 3 shows an example of the recovery ring of the primary server 00.

(3) Each recovery server, say server R, corresponds to a third layer ring called *backup ring* that is composed of the servers that are one-hop to R and two-hop to P. The backup copies of the objects of P are stored in the disks of backup servers on the backup rings. Fig. 3 shows an example of six backup rings.

In Fig. 3, all the 16 primary servers have the same

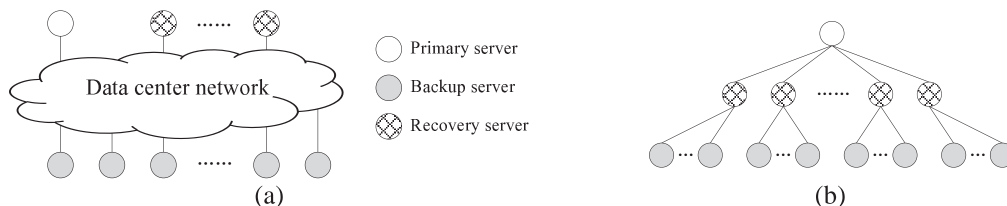

Figure 2: (a) Primary-recovery-backup structure [11]. (b) Directly connected tree in RAMCube.

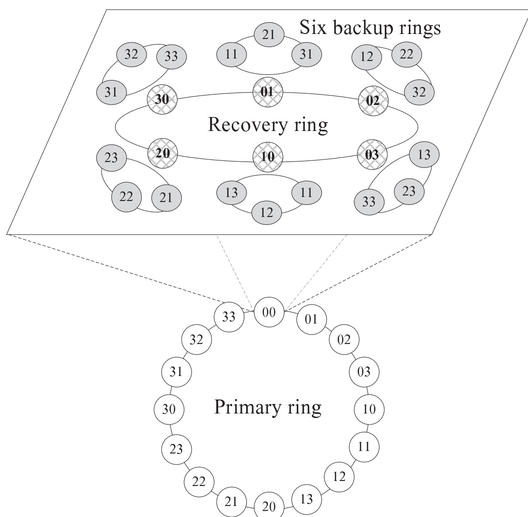

Figure 3: The primary ring of BCube$(4,1)$, and the recovery ring and backup rings of server 00.

primary-recovery-backup structure with server 00. In the symmetric MultiRing structure, if a server A is a primary/recovery/backup server of another server B, then B is also a primary/recovery/backup server of A. Consider a RAMCube constructed based on BCube(n,k). From the construction of primary/recovery/backup rings, and from the property of generalized Hypercube, we can see that there are n^{k+1}, $(n-1)(k+1)$, and $(n-1)k$ servers on the primary/recovery/backup ring, respectively. And a primary server has totally $\frac{(n-1)^2k(k+1)}{2}$ backup servers.

For BCube$(16,2)$, for example, there are 4096 primary servers, each of which has 45 recovery servers and 675 backup servers. Note that it is not mandatory for a primary server to employ all its recovery/backup servers. For example, a primary server in BCube$(16,2)$ may employ 30 (instead of 45) recovery servers on its recovery ring to reduce fragmentation, at the cost of longer recovery time and lower throughput.

For durability each object has multiple backup copies distributed in the backup servers' disks. Several factors should be considered for the placement of the backup copies. E.g., the copies should reside in different racks in case of rack power failure, and different disk I/O bandwidths should also be considered for balancing the load. In current RAMCube, the data of a primary server is partitioned and each partition is assigned to a recovery server, and the partition of a recovery server is divided into b sub-partitions each being assigned to its f backup servers, where b and f are respectively the number of servers on the backup ring and the replication factor.

3.3 Failure Detection and Recovery

Server failure. A primary server periodically sends heartbeats to each of its recovery servers. If a recovery server (say R) does not receive heartbeats from its primary server (say P) for a certain period, then R would confirm the failure of P by using BCube source routing (BSR) [9] to issue additional pings through k switches directly connected to P other than the one between R and P. E.g., in the network depicted in Fig. 1, if server 01 suspects server 00 fails since it cannot receive heartbeats from 00, it would send a ping message to 00 through switch $\langle 1,0 \rangle$ with a specified path $01 \rightarrow 11 \rightarrow 10 \rightarrow 00$.

If the additional pings also fail then R would independently start the recovery. In case of false failure detection due to rare conditions, e.g., all $k+1$ paths between P and R are *simultaneously* temporarily congested, relevant backup servers directly connected to R would reject any further backup requests from P and indicate P to stop servicing the corresponding sub key space. Therefore, in RAMCube false failure detection is NOT fatal but "expensive". Our local detection mechanism eliminates much of the possibility of false positive induced by network problems and thus reduces unnecessary recoveries.

During the recovery of a server failure, three cases have to be considered corresponding to the three roles of the failed server. For simplicity here we assume a primary server employs all its recovery servers.

(1) *Primary server failure*: Among multiple backup copies of each object, we assign one copy as the *dominant* copy. During recovery, the relevant recovery servers fetch dominant copies to their RAM from directly connected backup servers. Since the backup servers have exactly two digits different from their primary servers, each backup server services *two* recovery servers. E.g., if a primary server (say 00) fails in the network depicted in Fig. 1, a backup server (11) services two recovery servers (01 and 10). Given the normal configuration with 24∼64 GB data per primary server, 10 Gbps network

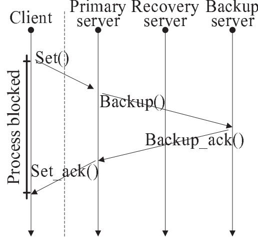

Figure 4: Processing a write request in RAMCube.

bandwidth and 100 MB/s disk bandwidth, a RAMCube constructed based on BCube$(16,2)$ can easily recover a primary server failure in 1∼2 seconds.

(2) *Recovery server failure*: For each involved primary server, the affected backup servers first register to the other heathy recovery server . Then, RAMCube moves backup data to other backup servers connecting to the registered healthy recovery servers. For example, in the network depicted in Fig. 1, if the recovery server (01) of a primary server (00) fails, RAMCube first registers a new recovery server for each affected backup server (10 for 11, 20 for 21, and 30 for 31); and then moves backup data to the newly chosen backup servers on the same backup rings (11→(12,13), 21→(22,23), and 31→(32,33)).

(3) *Backup server failure*: RAMCube simply copies affected data to healthy servers on the same backup ring.

Switch failure. RAMCube can easily handle switch failures. For example, in the network depicted in Fig. 1, if servers 01, 02, 03 all find server 00 is unreachable through switch $\langle 0,0 \rangle$ but they can receive ping acknowledgements through switch $\langle 1,0 \rangle$, then RAMCube considers switch $\langle 0,0 \rangle$ failed. Since a switch failure in BCube results in only graceful performance degradation [9] but no data loss or unavailability, it is not critical and we can replace the failed switch in a relatively longer period of time (compared to server failures).

4 Prototype Implementation & Evaluation

We have prototyped RAMCube by implementing a user-level service in Windows Server 2008 R2, which contains a *connection manager* and a *memory manager*. The *connection manager* maintains the status of directly connected neighbors and handles network events including receiving data from clients and sending/receiving backup/recovery data. The *memory manager* uses a slab-based mechanism and handles *set/get/delete* requests inside a server. It also uses a simple log-structured approach similar to previous logging file systems [12, 11] for asynchronously writing backup data (divided into 8 MB segments) into the disks of backup servers.

When the primary server receives a key-value pair, it stores the data in its RAM (handled by the *memory manager*) and sends a copy to each of the relevant backup servers simultaneously. The backup servers return an acknowledgement after the copies are written to the RAM (not disks), and then the primary server acknowledges to its client that the write operation is successful. This approach provides efficiency as well as simplicity both in normal read/write operations and in failure recovery. This procedure is depicted in Fig. 4.

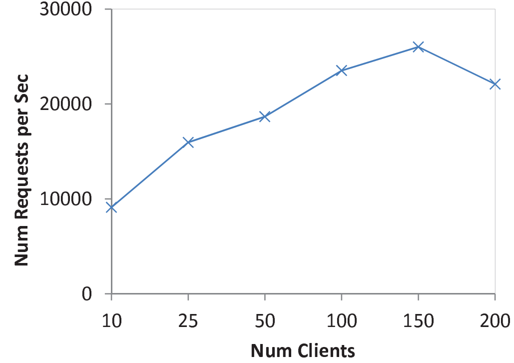

Figure 5: RAMCube server throughput.

We have built a RAMCube testbed with 16 Dell R610 servers and 8 8-port DLink Gigabit Ethernet switches constructing a BCube$(4,1)$ network. Each server has one Intel Xeon 2.27 GHz quad core CPU and 16 GB RAM, and installs one Seagate ST9500430SS 7200 RPM, 1 TB disk. Since the ServerSwitch 10 Gbps NIC is still under development, we install one ServerSwitch 1 Gbps NIC [10] on each server.

For simplicity, the RAMCube benchmark is a single-threaded busy loop (running on a client machine) where many clients asynchronously communicate with the RAMCube server, which is also single-threaded.

Our first experiment evaluates the throughput of RAMCube, measured by the number of requests handled by a primary server per second. The client performs write operations with the form of "*set* key value" and the primary server indicates the success of each write with an acknowledgement. The object size is 100 bytes. We measure the number of *set* requests finished per second as a function of the number of clients in the busy loop. The result (depicted in Fig. 5) shows the maximum throughput of RAMCube on one core is about 26,000 writes/sec. Currently the throughput of RAMCube is affected by a series of factors such as the overhead of memory manager and connection manager at primary servers and the logging performance at backup servers. We can improve this by using more cores at primary servers and installing more disks at backup servers. We also measures the throughput of memcached. Its maximum throughput is about 40,000 writes/sec.

In our second experiment we first fill a primary server with 2 GB data (each object having 100 bytes), and then

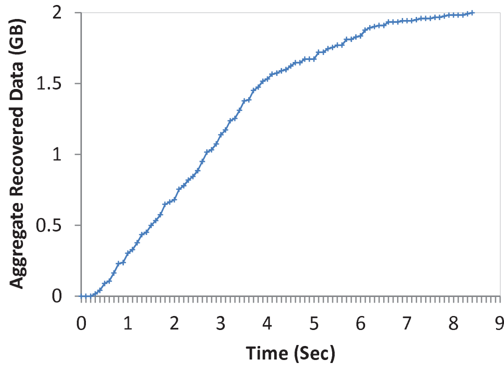

Figure 6: RAMCube recovery time.

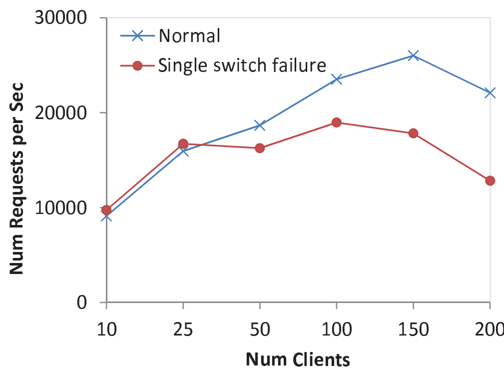

Figure 7: RAMCube throughput with a single switch failure.

cause a failure of that server and measure the aggregate recovered data size (of all six recovery servers) over time. The heartbeat timeout is set to 300 ms. The result is depicted in Fig. 6. The minimum and maximum recovery time of six recovery servers are respectively 3.5 and 8.3 seconds. This difference shows that although RAMCube removes the congestion of multi-hop traffic in recovery, we still have a lot of work to do for scheduling the one-hop burst recovery traffic. The maximum aggregate recovery bandwidth is about 456 MB/sec, which is limited by the NIC bandwidth and the number of recovery servers. We can expect a much faster recovery speed by using ServerSwitch 10G NIC and more recovery/backup servers in a larger RAMCube with more levels (NIC ports).

Previous designs, e.g., RAMCloud [11], cannot handle switch failures well due to overwhelming recovery traffic congestion. This is a critical problem that prevents RAM-based storage systems from being practical. We evaluate the throughput of RAMCube with a single switch failure. Results (depicted in Fig. 7) show RAMCube has a graceful performance degradation during single switch failures.

5 Conclusion

We have presented the design of RAMCube as a novel RAM-based key-value store for high-performance I/O in data center networks. By exploiting the proximity of BCube network, RAMCube restricts all failure detection and recovery traffic within one-hop neighborhood.

Acknowledgement

This work was supported in part by the National Basic Research Program of China (973) under Grant No. 2011CB302601, the National Natural Science Foundation of China (NSFC) under Grant No. 60903205 and 61003076, and the Doctoral Program of Higher Education under Grant No. 20094307110008.

References

[1] http://memcached.org/.

[2] Bhuyan, L. N., and Agrawal, D. P. Generalized hypercube and hyperbus structures for a computer network. *IEEE Trans. Computers 33*, 4 (1984), 323–333.

[3] Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. The primary-backup approach, 1993.

[4] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. Bigtable: A distributed storage system for structured data. In *OSDI* (2006), pp. 205–218.

[5] Daniels, D., Doo, L. B., Downing, A., Elsbernd, C., Hallmark, G., Jain, S., Jenkins, B., Lim, P., Smith, G., Souder, B., and Stamos, J. Oracle's symmetric replication technology and implications for application design. In *SIGMOD* (1994), ACM Press, p. 467.

[6] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *SOSP* (2007), pp. 205–220.

[7] Garcia-Molina, H., and Salem, K. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng. 4*, 6 (1992), 509–516.

[8] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[9] Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., and Lu, S. Bcube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM* (2009), pp. 63–74.

[10] Lu, G., Guo, C., Li, Y., Zhou, Z., Yuan, T., Wu, H., Xiong, Y., Gao, R., and Zhang, Y. Serverswitch: A programmable and high performance platform for data center networks. In *NSDI* (2011).

[11] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J. K., and Rosenblum, M. Fast crash recovery in ramcloud. In *SOSP* (2011), pp. 29–41.

[12] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. In *SOSP* (1991), pp. 1–15.

[13] Zhang, Y., Li, D., Chu, R., Xiao, N., and Lu, X. Pibus: A network memory-based peer-to-peer io buffering service. In *Networking* (2007), pp. 1237–1240.